
https://algs4.cs.princeton.edu

# Dynamic Programming

- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*
- *shortest paths in digraphs*


https://algs4.cs.princeton.edu

# Dynamic Programming

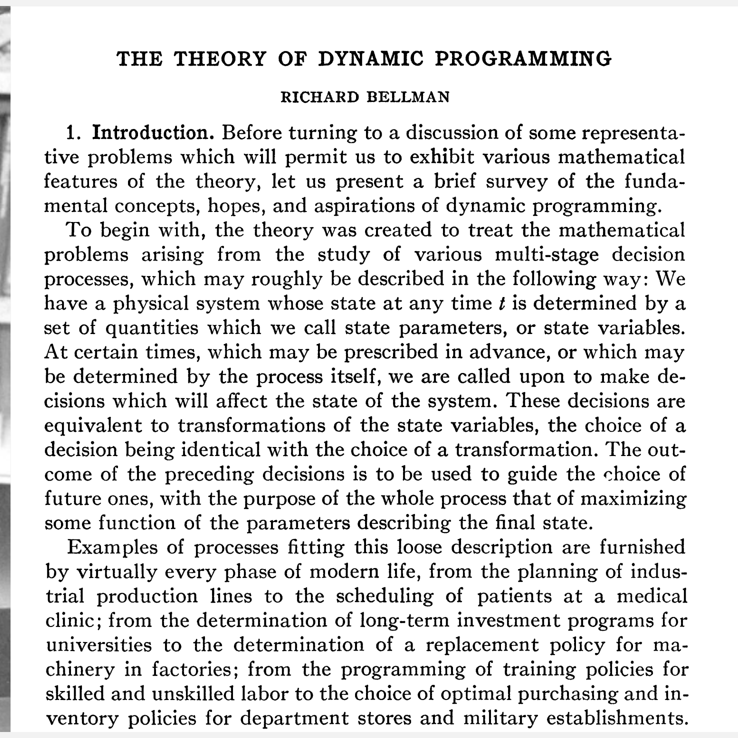
- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*
- *shortest paths in digraphs*

# Dynamic programming

Algorithm design paradigm.

- Break up a problem into a series of overlapping subproblems.
- Build up solutions to larger and larger subproblems.

  (caching solutions to subproblems for later reuse)



THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time $t$ is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

**Richard Bellman, *46**

Application areas.

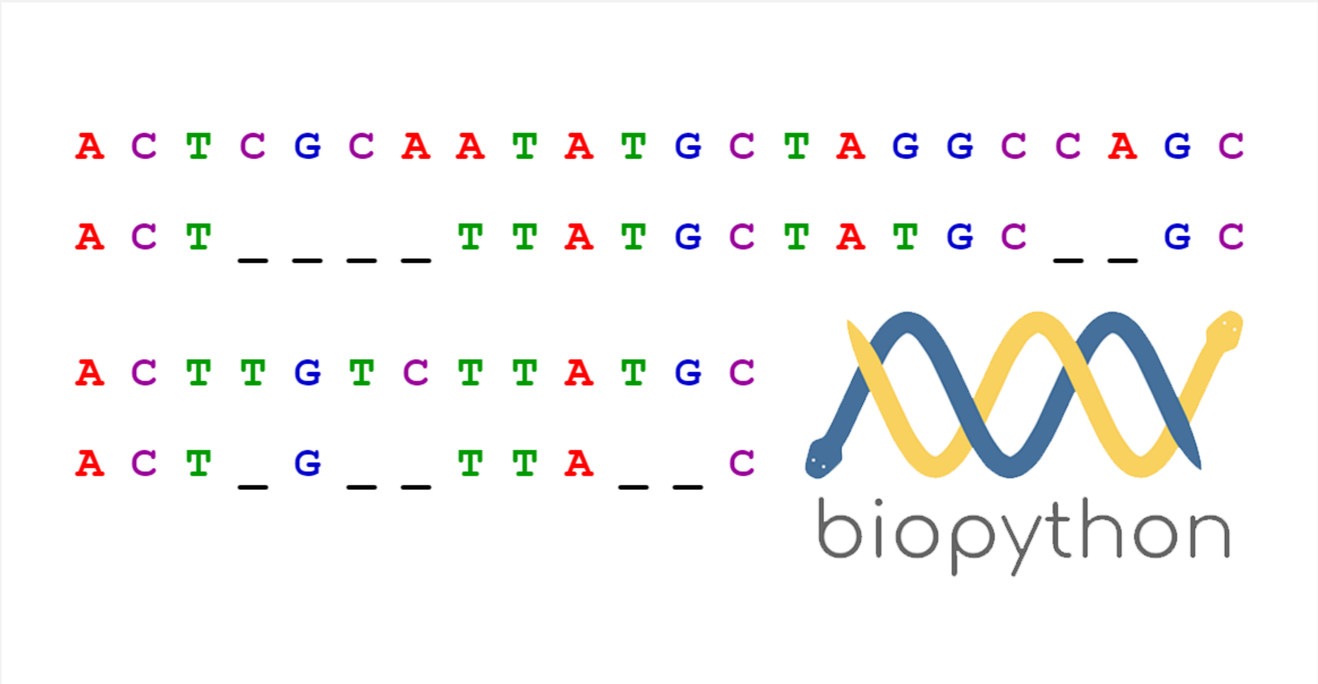
- Operations research: multistage decision processes, control theory, optimization, ...
- Computer science: AI, compilers, systems, graphics, theory, ....
- Economics.
- Bioinformatics.
- Information theory.
- Tech job interviews.

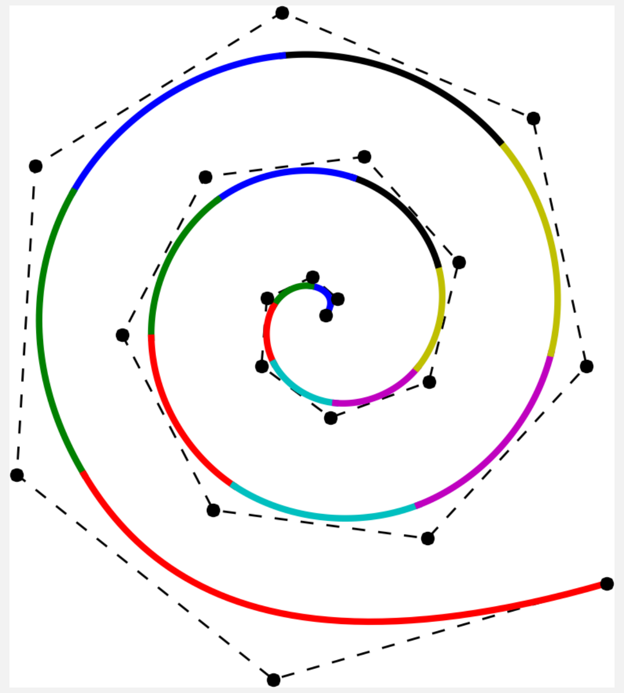
Bottom line. Powerful technique; broadly applicable.

# Dynamic programming algorithms

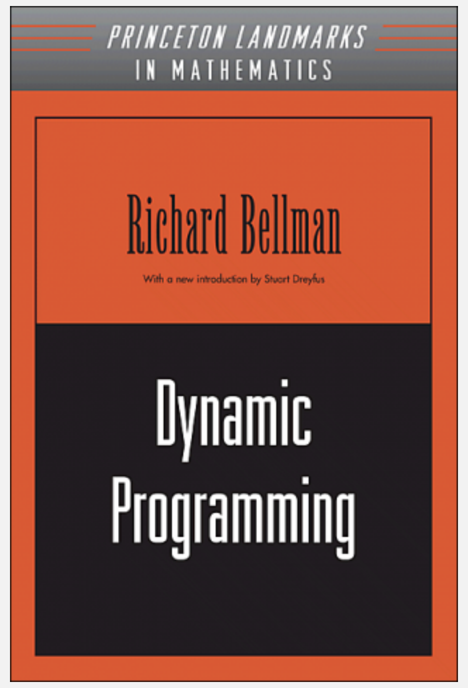
Some famous examples.

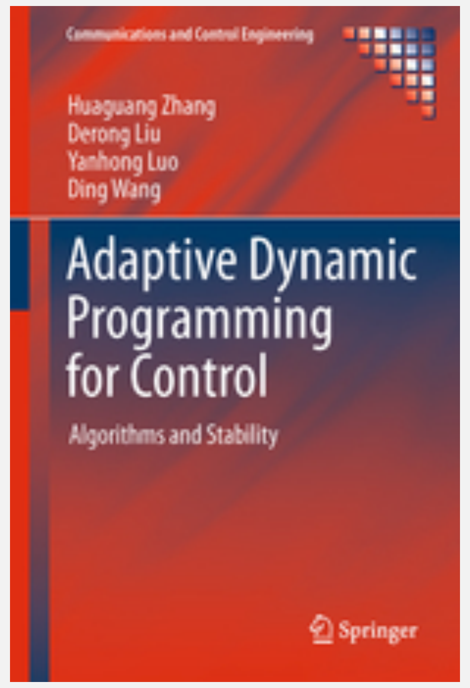
- Needleman–Wunsch/Smith–Waterman for sequence alignment.
- Cocke–Kasami–Younger for parsing context-free grammars.
- Knuth–Plass for word wrapping text in $\TeX$.
- Bellman–Ford–Moore for shortest path.
- De Boor for evaluating spline curves.
- Viterbi for hidden Markov models.
- Unix diff for comparing two files.
- Avidan–Shamir for seam carving.
- NP-hard graph problems on trees (vertex color, vertex cover, independent set, ...).
- ...

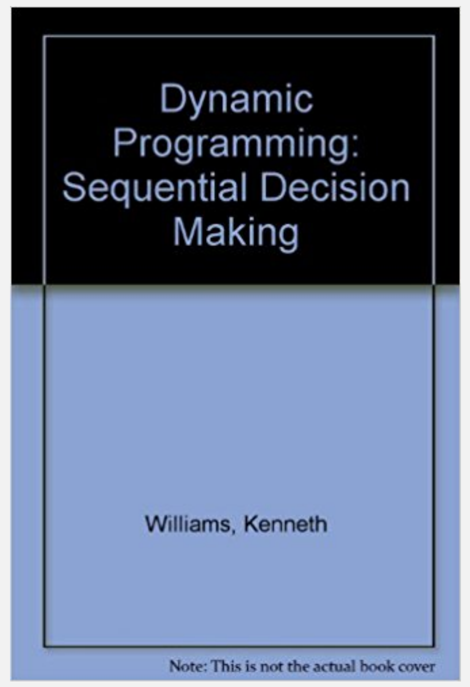


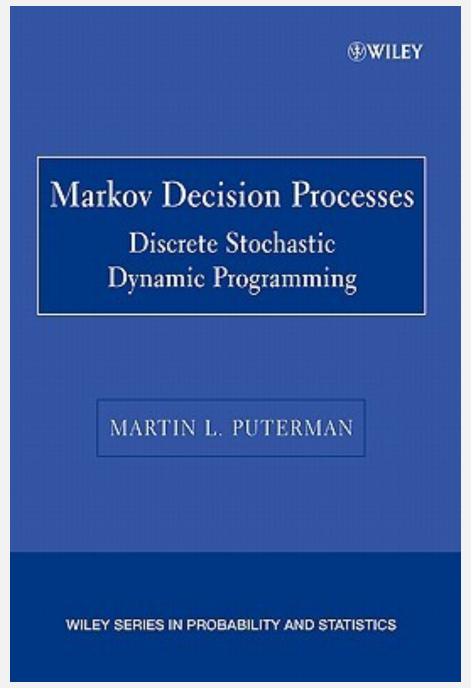



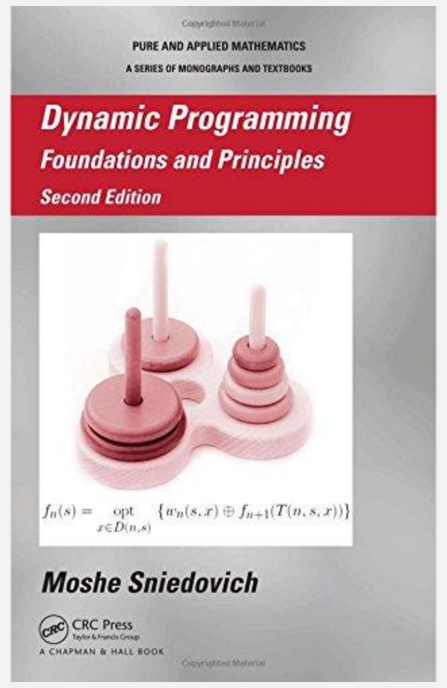
# Dynamic programming books

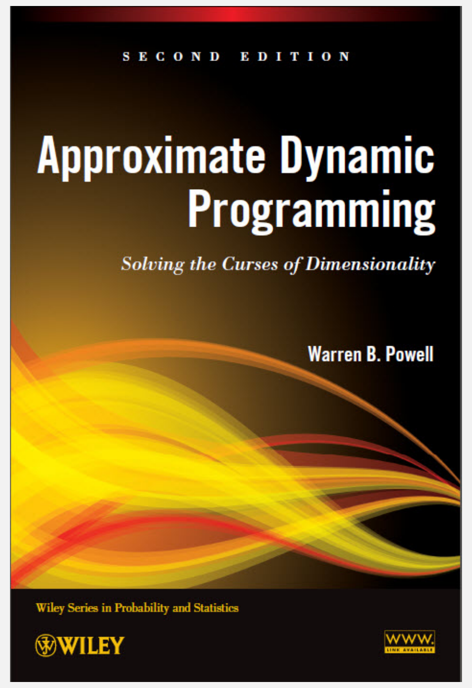


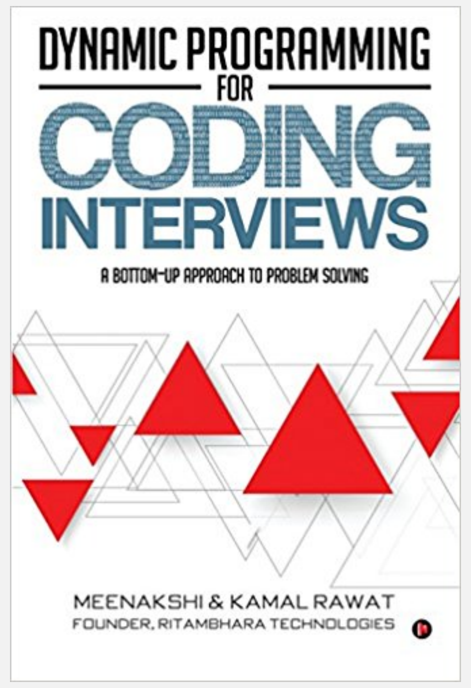



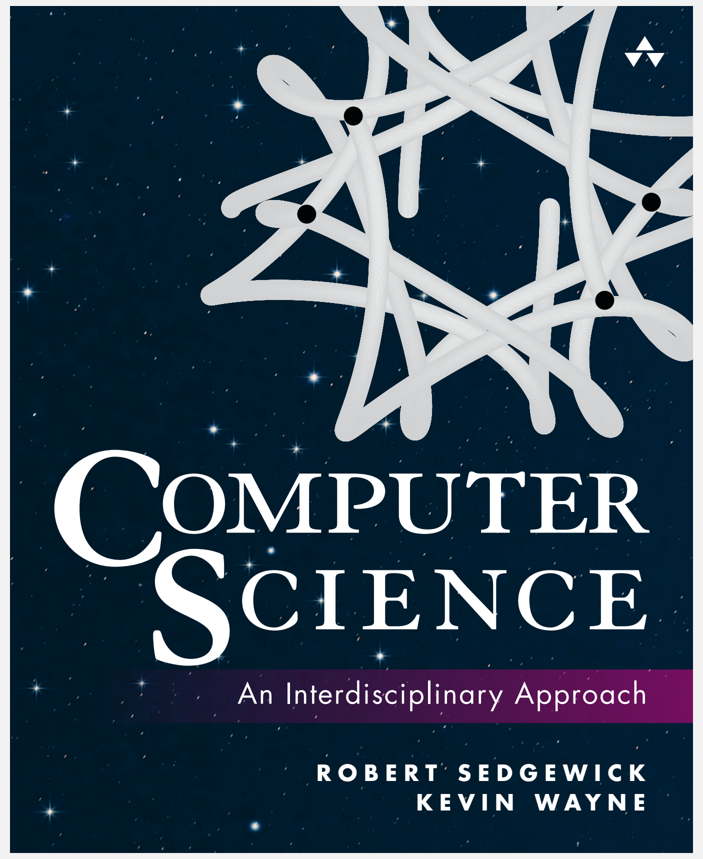



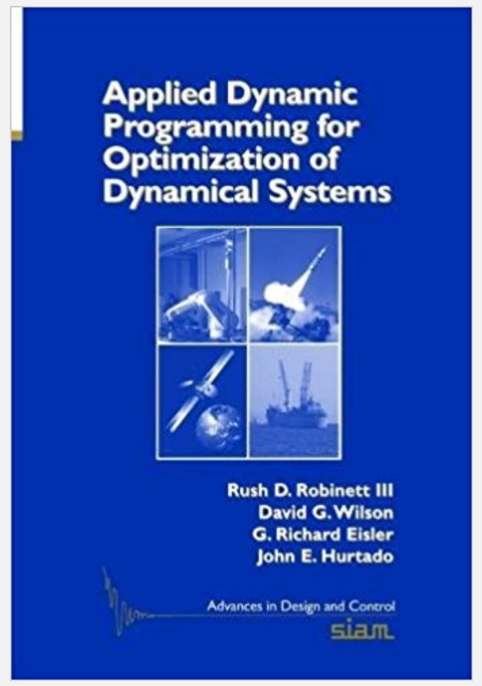



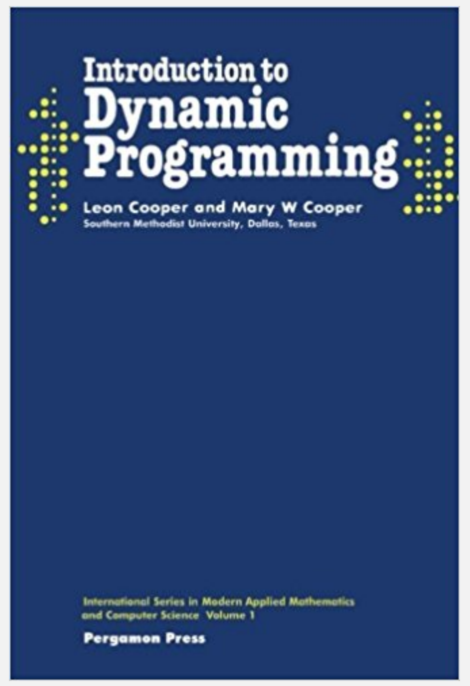



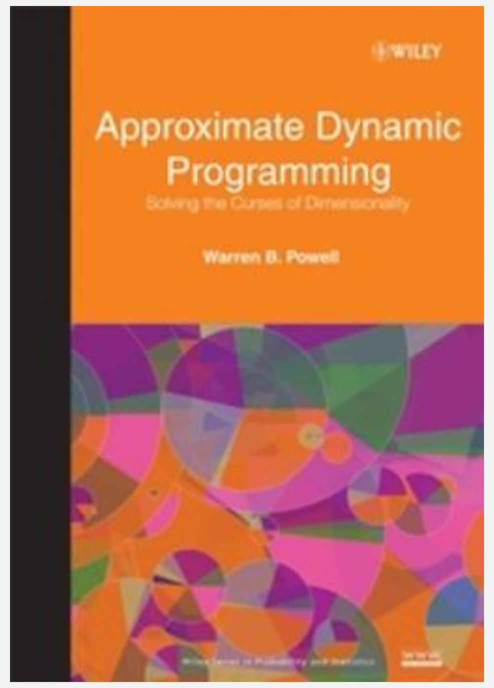



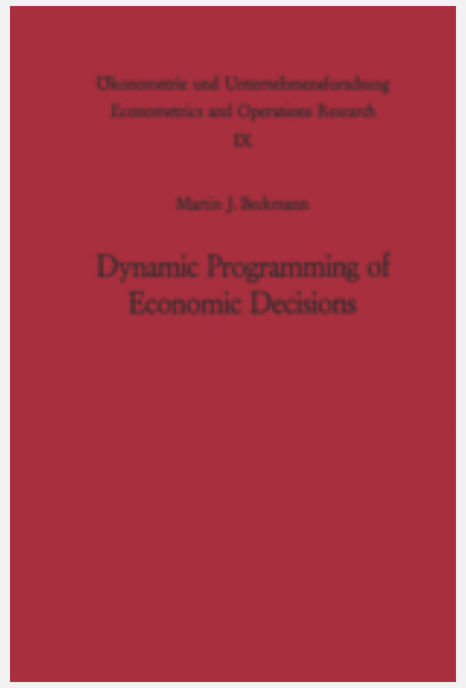



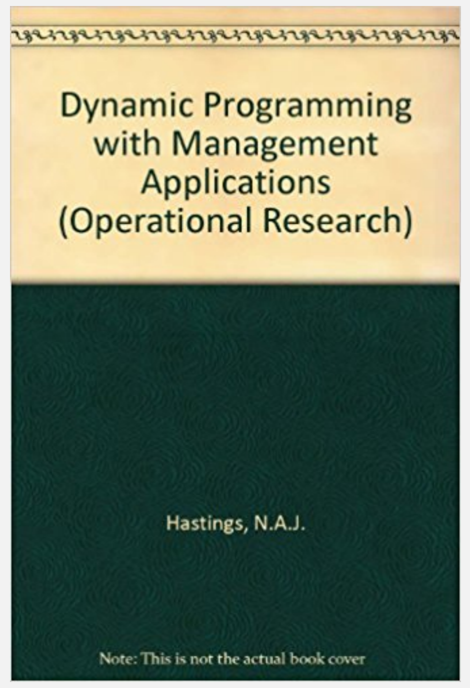

pp. 284–289

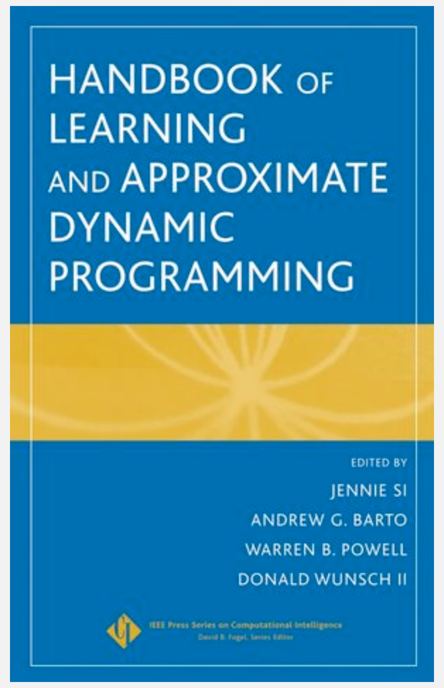


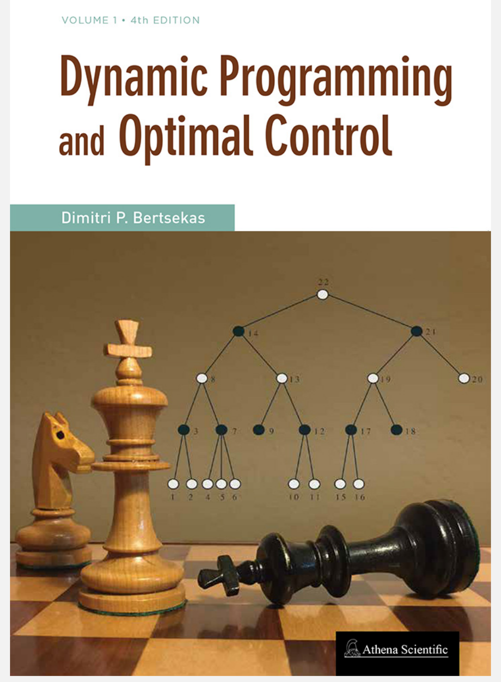



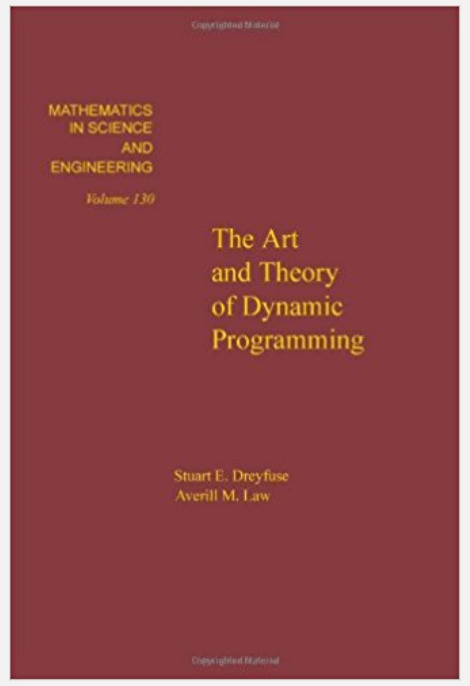



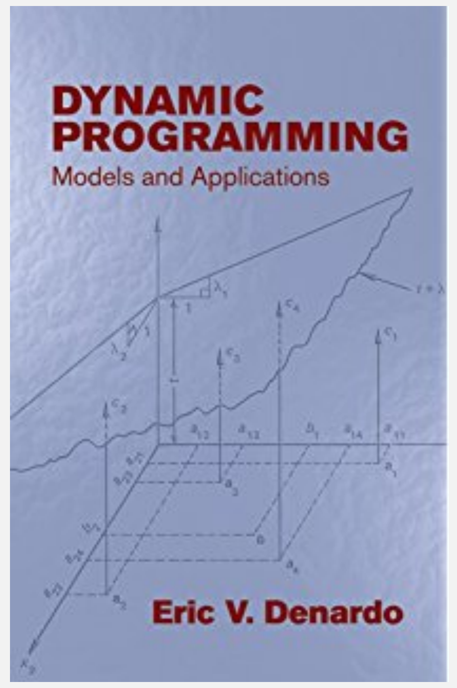



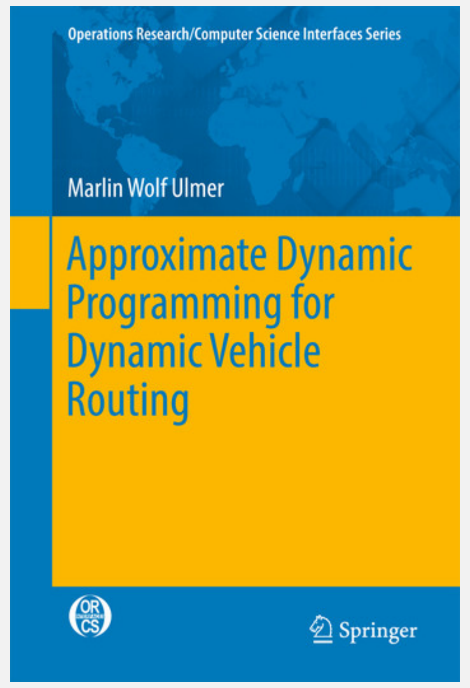



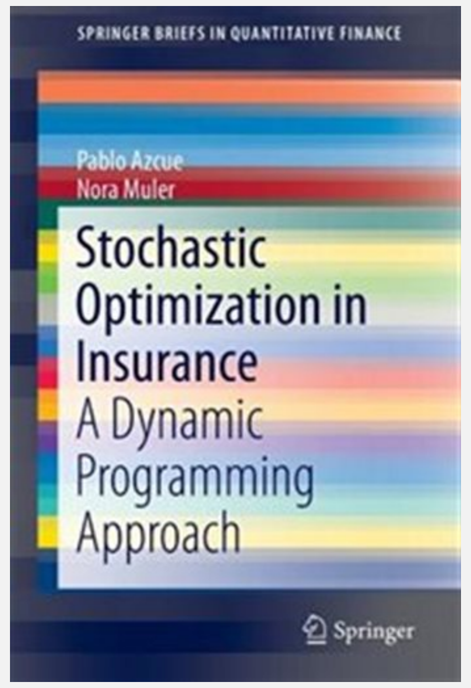



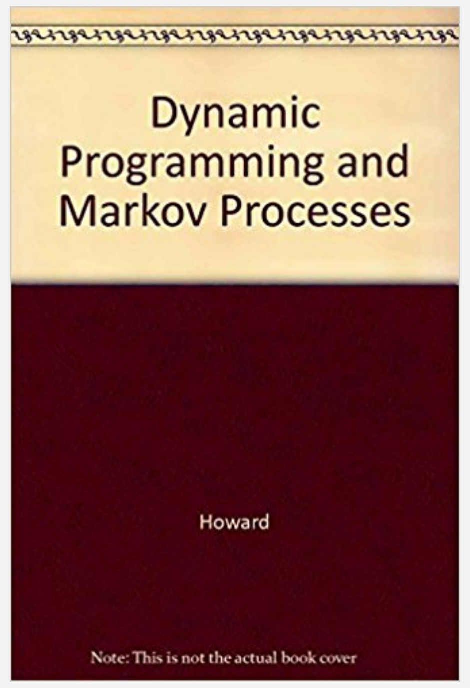



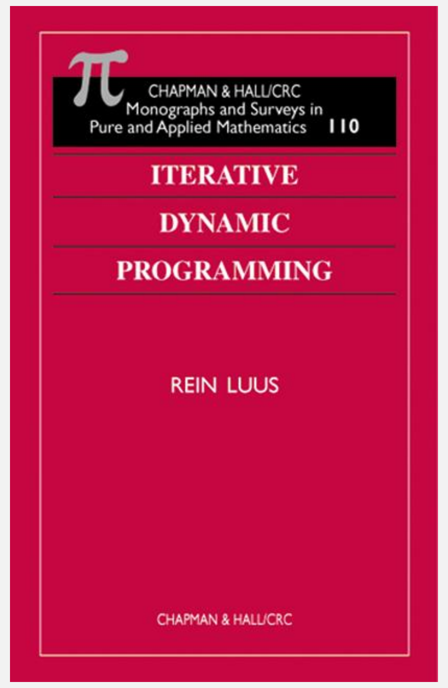



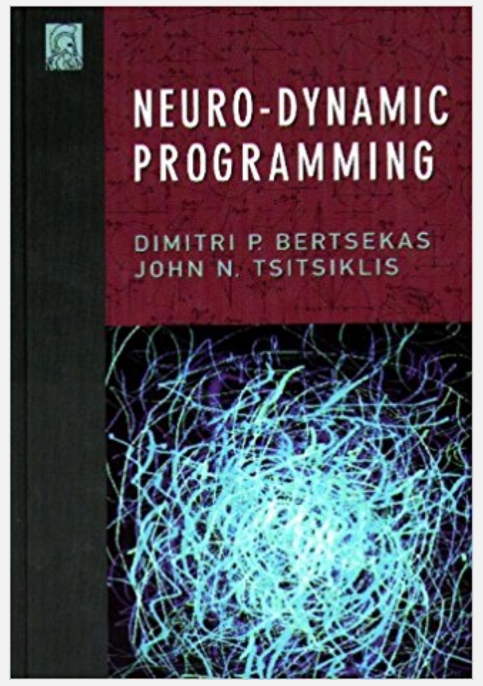




https://algs4.cs.princeton.edu

# Dynamic Programming

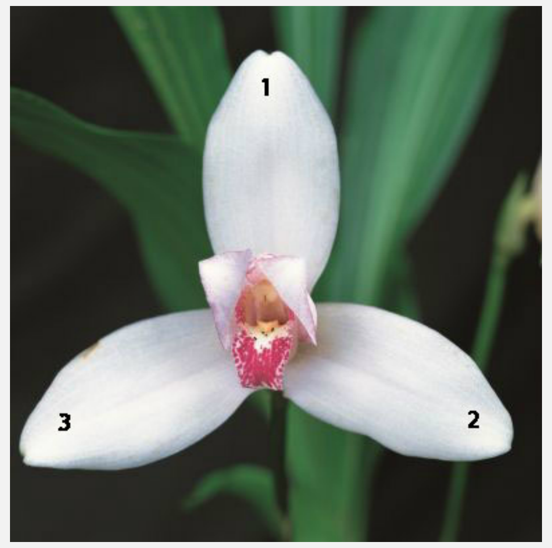
- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*
- *shortest paths in digraphs*

# Fibonacci numbers

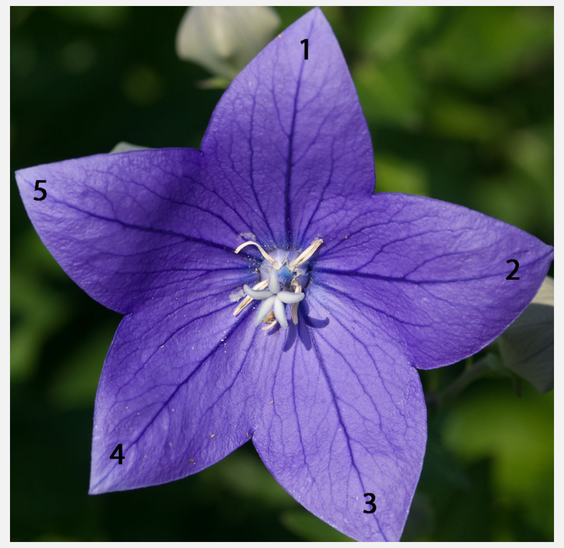
Fibonacci numbers. $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ \ldots$

$$F_i = \begin{cases} 0 & \text{if } \ i = 0 \\ 1 & \text{if } \ i = 1 \\ F_{i-1} + F_{i-2} & \text{if } \ i > 1 \end{cases}$$

**Leonardo Fibonacci**



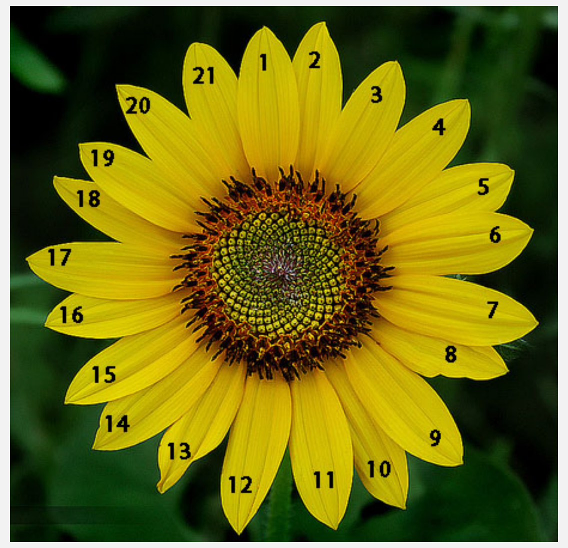

**3**




**5**


**8**

**13**




**21**

**34**

**55**

**89**

# Fibonacci numbers: naïve recursive approach

**Fibonacci numbers.** $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

$$F_i = \begin{cases} 0 & \text{if } \ i = 0 \\ 1 & \text{if } \ i = 1 \\ F_{i-1} + F_{i-2} & \text{if } \ i > 1 \end{cases}$$

**Goal.** Given $n$, compute $F_n$.

**Naïve recursive approach:**

```
public static long fib(int i)
{
   if (i == 0) return 0;
   if (i == 1) return 1;
   return fib(i-1) + fib(i-2);
}
```

**How long to compute fib(75) using the naïve recursive algorithm?**

**A.** Less than 1 second.

**B.** 1 minute.

**C.** More than 1 year.

**D.** Result won't fit in a 64-bit `long` integer.

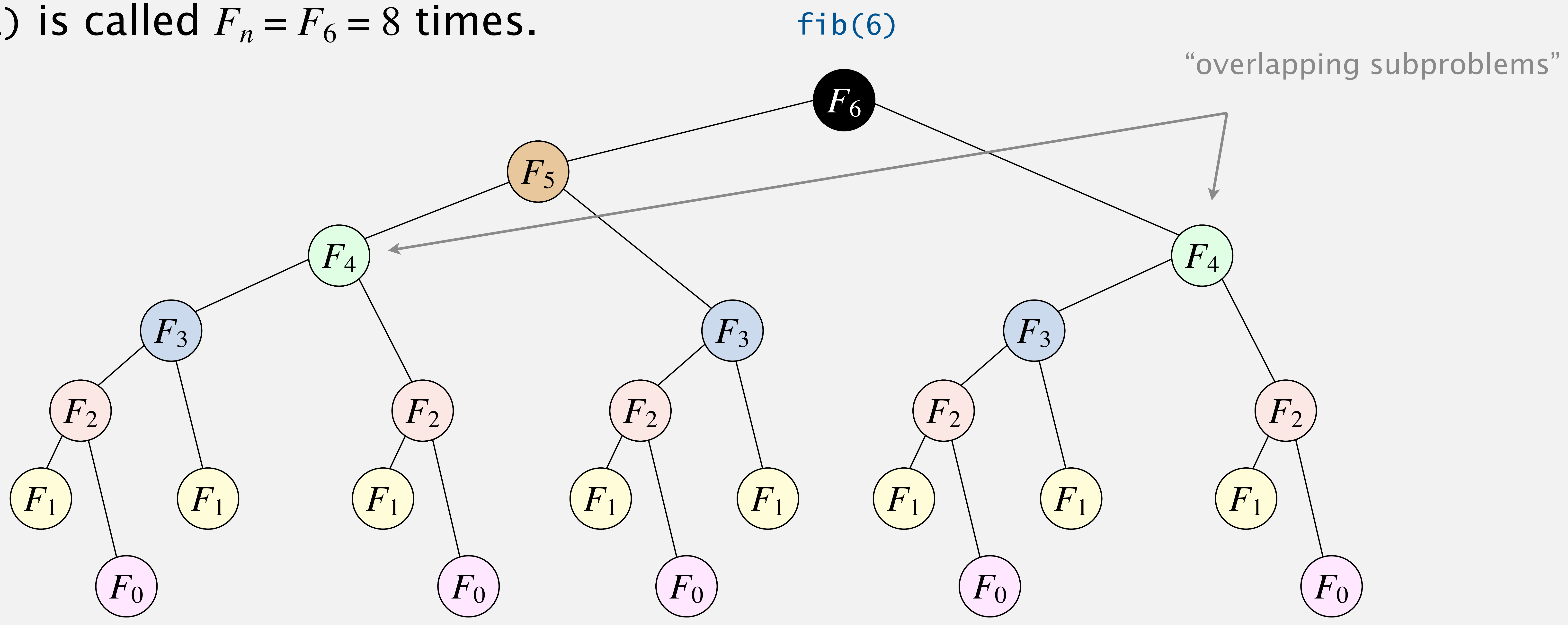
# Fibonacci numbers: recursion tree and exponential growth

Exponential waste. Same overlapping subproblems are solved repeatedly.

Ex. To compute `fib(6)`:

- `fib(5)` is called 1 time.
- `fib(4)` is called 2 times.
- `fib(3)` is called 3 times.
- `fib(2)` is called 5 times.
- `fib(1)` is called $F_n = F_6 = 8$ times.

$$F_n \sim \phi^n, \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$




**running time = # subproblems × cost per subproblem**

# Fibonacci numbers: top-down dynamic programming

**Memoization.**

- Maintain an array (or symbol table) to remember all computed values.
- If value to compute is known, just return it; otherwise, compute it; remember it; and return it.

```
public static long fib(int i)
{
   if (i == 0) return 0;
   if (i == 1) return 1;
   if (f[i] == 0) f[i] = fib(i-1) + fib(i-2);
   return f[i];
}
```

assume global long array f[], initialized to 0

**Impact.** Solves each subproblem $F_i$ only once; $\Theta(n)$ time to compute $F_n$.

# Fibonacci numbers: bottom-up dynamic programming

Bottom-up dynamic programming.

- Build computation from the "bottom up."
- Solve small subproblems and save solutions.
- Use those solutions to solve larger subproblems.

```
public static long fib(int n)
{
   long[] f = new long[n+1];
   f[0] = 0;
   f[1] = 1;
   for (int i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

smaller subproblems

Impact. Solves each subproblem $F_i$ only once; $\Theta(n)$ time to compute $F_n$; no recursion.

# Fibonacci numbers: further improvements

## Performance improvements.

- Save space by saving only two most recent Fibonacci numbers.

```
public static long fib(int n) {
   int f = 1, g = 0;
   for (int i = 1; i < n-1; i++) {
      f = f + g;
      g = f - g;
   }
   return f;
}
```

f and g are consecutive Fibonacci numbers

- Exploit additional properties of problem:

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} \right\rceil, \quad \phi = \frac{1+\sqrt{5}}{2}$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}^n$$

# Dynamic programming recap

Dynamic programming.

- Divide a complex problem into a number of simpler overlapping subproblems.

  (define $n + 1$ subproblems, where subproblem $i$ is computing the $i^{th}$ Fibonacci number)

- Define a recurrence relation to solve larger subproblems from smaller subproblems.

  (easy to solve subproblem $i$ if we know solutions to subproblems $i - 1$ and $i - 2$)

$$F_i = \begin{cases} 0 & \text{if } \; i = 0 \\ 1 & \text{if } \; i = 1 \\ F_{i-1} + F_{i-2} & \text{if } \; i > 1 \end{cases}$$

- Store solutions to each of these subproblems, solving each subproblem only once.

  (use an array `fib[i]` to store solution to subproblem $i$)

- Use stored solutions to solve the original problem.

  (subproblem $n$ is original problem)

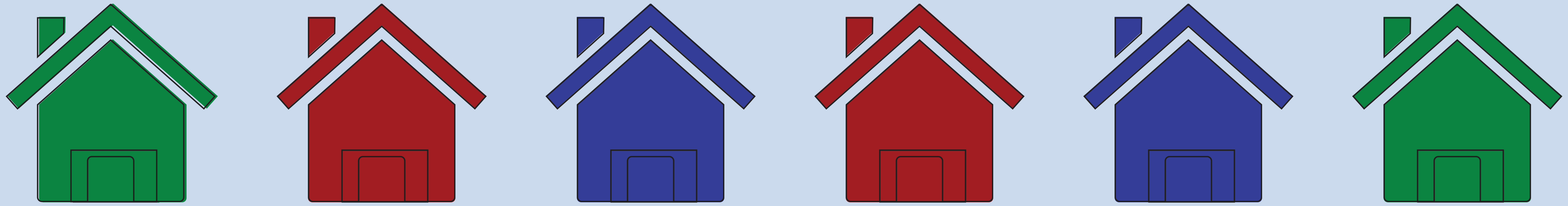

https://algs4.cs.princeton.edu

# Dynamic Programming

# House coloring problem

Goal. Paint a row of $n$ houses red, green, or blue so that:

- Minimize total cost, where $cost(i, color)$ is cost to paint $i$ given color.
- No two adjacent houses have the same color.



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $cost(i, red)$ | 7 | 6 | 7 | 8 | 9 | 20 |
| $cost(i, green)$ | 3 | 8 | 9 | 22 | 12 | 8 |
| $cost(i, blue)$ | 16 | 10 | 4 | 2 | 5 | 7 |

**cost to paint house i the given color**
**(3 + 6 + 4 + 8 + 5 + 8 = 34)**

# HOUSE COLORING PROBLEM: DYNAMIC PROGRAMMING FORMULATION

Goal. Paint a row of $n$ houses red, green, or blue so that:

- Minimize total cost, where $cost(i, color)$ is cost to paint $i$ given color.
- No two adjacent houses have the same color.

Subproblems.

- $R(i)$ = min cost to paint houses $1, \ldots, i$ with $i$ red.
- $G(i)$ = min cost to paint houses $1, \ldots, i$ with $i$ green.
- $B(i)$ = min cost to paint houses $1, \ldots, i$ with $i$ blue.
- Optimal cost = $\min \{ R(n), G(n), B(n) \}$.

Dynamic programming recurrence.

- $R(i) = cost(i, red) + \min \{ G(i-1), B(i-1) \}$
- $G(i) = cost(i, green) + \min \{ B(i-1), R(i-1) \}$
- $B(i) = cost(i, blue) + \min \{ R(i-1), G(i-1) \}$

"optimal substructure"
(optimal solution can be constructed from optimal solutions to smaller subproblems)

# House coloring: naïve recursive implementation

A mutually recursive implementation.

```
private int red(int i) {
   if (i == 0) return 0;
   return cost[i][RED] + Math.min(green(i-1), blue(i-1));
}

private int green(int i) {
   if (i == 0) return 0;
   return cost[i][GREEN] + Math.min(red(i-1), blue(i-1));
}

private int blue(int i) {
   if (i == 0) return 0;
   return cost[i][BLUE] + Math.min(red(i-1), green(i-1));
}

public int optimalValue() {
   return min3(red(n), blue(n), green(n));
}
```

$$R(i) \;=\; cost(i, red) \;+\; \min \{\, G(i-1),\; B(i-1) \,\}$$

$$G(i) \;=\; cost(i, green) + \min \{\, B(i-1),\; R(i-1) \,\}$$

$$B(i) \;=\; cost(i, blue) \;+\; \min \{\, R(i-1),\; G(i-1) \,\}$$

# Dynamic programming: quiz 2

**What is running time of the naïve recursive algorithm as a function of n?**

**A.** $\Theta(n)$

**B.** $\Theta(n^2)$

**C.** $\Theta(2^n)$

**D.** $\Theta(n!)$

## Dynamic programming aphorism

> *" Those who cannot remember the past are condemned to repeat it. "*
>
> **— *Dynamic Programming***
>
> (Jorge Agustín Nicolás Ruiz de Santayana y Borrás)

# House coloring: bottom-up implementation

Bottom-up implementation.

```
int[] r = new int[n+1];
int[] g = new int[n+1];
int[] b = new int[n+1];

for (int i = 1; i <= n; i++) {
   r[i] = cost[i][RED]   + Math.min(g[i-1], b[i-1]);
   g[i] = cost[i][GREEN] + Math.min(b[i-1], r[i-1]);
   b[i] = cost[i][BLUE]  + Math.min(r[i-1], g[i-1]);
}

return min3(r[n], g[n], b[n]);
```

$$R(i) = cost(i, red) + \min\{ G(i-1), B(i-1) \}$$

$$G(i) = cost(i, green) + \min\{ B(i-1), R(i-1) \}$$

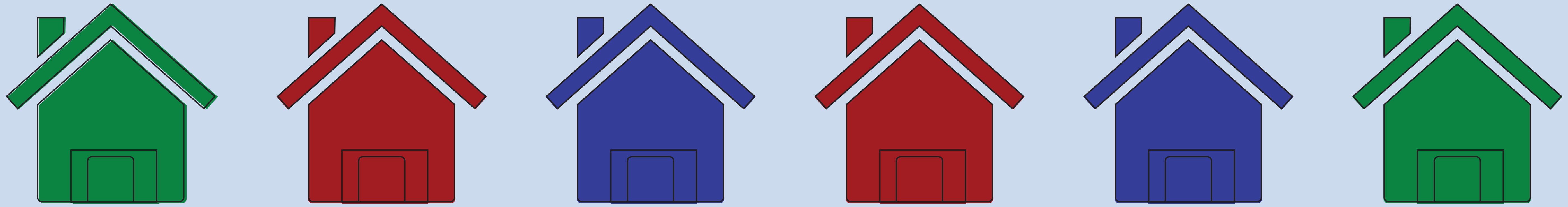
$$B(i) = cost(i, blue) + \min\{ R(i-1), G(i-1) \}$$

Proposition. The bottom-up DP algorithm takes $\Theta(n)$ time.

# House coloring: reconstructing the solution (backtrace)

So far: we've computed the value of the optimal solution.

Still need: the solution itself (which color to paint each house).



| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $R[i]$ | 7 | 9 | 20 | 21 | 29 | 46 |
| $G[i]$ | 3 | 15 | 18 | 35 | 32 | 34 |
| $B[i]$ | 16 | 13 | 13 | 20 | 26 | 36 |

**cost to paint houses 1, 2, …, i with house i the given color**

# COIN CHANGING

Problem. Given $n$ coin denominations $\{ d_1, d_2, \ldots, d_n \}$ and a target value $V$, find the fewest coins needed to make change for $V$ (or report impossible).

vending machine
(out of nickels)

Ex. Coin denominations = $\{1, 10, 25, 100\}$, $V = 130$.

Greedy (8 coins). $131¢ = 100 + 25 + 1 + 1 + 1 + 1 + 1 + 1.$

Optimal (5 coins). $131¢ = 100 + 10 + 10 + 10 + 1.$

8 coins
(131¢)

5 coins
(131¢)

Useful fact. Greedy algorithm is optimal for U.S. coin denominations $\{1, 5, 10, 25, 100\}$.

# Coin changing: dynamic programming formulation

Problem. Given $n$ coin denominations $\{ d_1, d_2, \ldots, d_n \}$ and a target value $V$, find the fewest coins needed to make change for $V$ (or report impossible).

Subproblems. $OPT(v)$ = fewest coins needed to make change for amount $v$.

Optimal value. $OPT(V)$.

Multiway choice. To compute $OPT(v)$,

- Select a coin of denomination $d_i \le v$ for some $i$.
- Use fewest coins to make change for $v - d_i$.

take best

optimal substructure

Dynamic programming recurrence.

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{ 1 + OPT(v - d_i) \} & \text{if } v > 0 \end{cases}$$

# COIN CHANGING: BOTTOM-UP IMPLEMENTATION

Bottom-up DP implementation.

```
int[] opt = new int[V+1];
for (int v = 1; v <= V; v++)
{
   opt[v] = Integer.MAX_VALUE;
   for (int i = 1; i <= n; i++)
   {
      if (d[i] > v) continue;
      if (opt[v] > 1 + opt[v - d[i]])
          opt[v] = 1 + opt[v - d[i]];
   }
}
```

$$OPT(v) = \begin{cases} \infty & \text{if } v < 0 \\ 0 & \text{if } v = 0 \\ \min\limits_{1 \le i \le n} \{ \, 1 + OPT(v - d_i) \, \} & \text{if } v > 0 \end{cases}$$

Running time. The bottom-up DP algorithm takes $\Theta(n\ V)$ time.

Note. Not polynomial in input size (and no poly-time algorithm is known).

$n$, $\log V$


https://algs4.cs.princeton.edu

# Dynamic Programming

- *introduction*
- *Fibonacci numbers*
- *interview problems*
- *shortest paths in DAGs*
- *shortest paths in digraphs*

# Shortest paths in directed acyclic graphs: dynamic programming formulation

**Problem.** Given a DAG with positive edge weights, find shortest $s \rightsquigarrow v$ path for each vertex $v$.

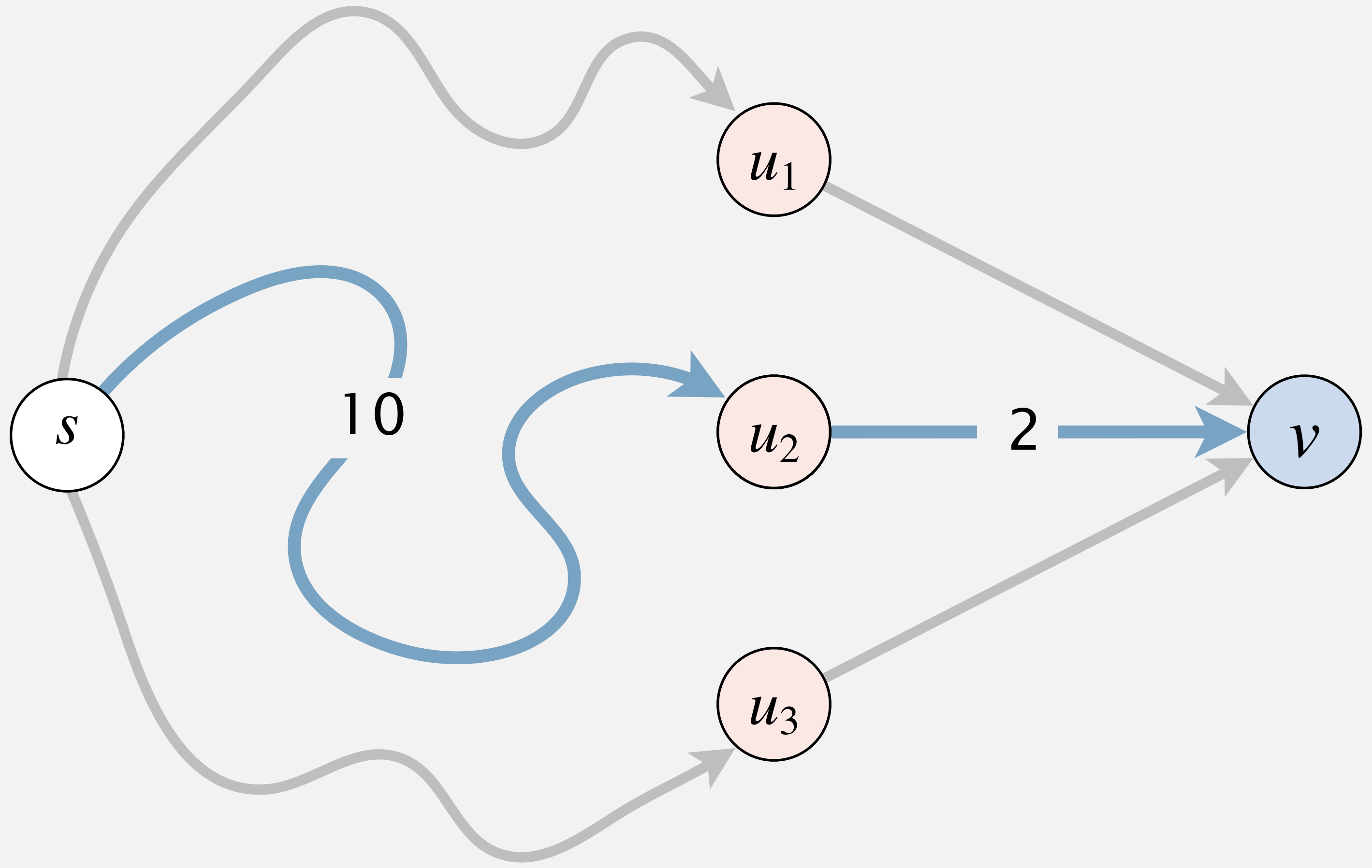
**Subproblems.** $distTo(v)$ = length of shortest $s \rightsquigarrow v$ path.

**Goal.** $distTo(v)$ for each $v$.

**Multiway choice.** To compute $distTo(v)$ :
- Select an edge $e = u \rightarrow v$ entering $v$.
- Combine with shortest $s \rightsquigarrow u$ path.

take best

optimal substructure




**Dynamic programming recurrence.**

$$distTo(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{e = u \rightarrow v} \{ distTo(u) + weight(e) \} & \text{if } v \neq s \end{cases}$$

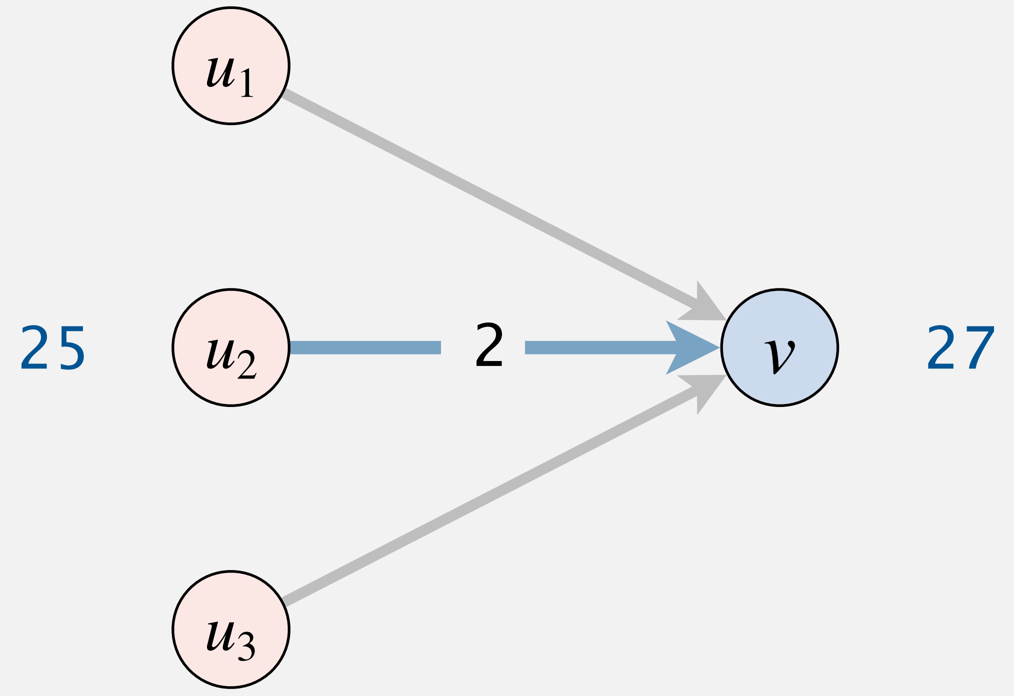
# Shortest paths in directed acyclic graphs: bottom-up solution

Bottom-up DP algorithm. Takes $\Theta(E + V)$ time and space with two tricks:

- Solve subproblems in topological order. ← ensures that "small" subproblems are solved before "large" ones
- Form reverse digraph $G^R$ to iterate over edges incident to vertex $v$.

Finding shortest paths.

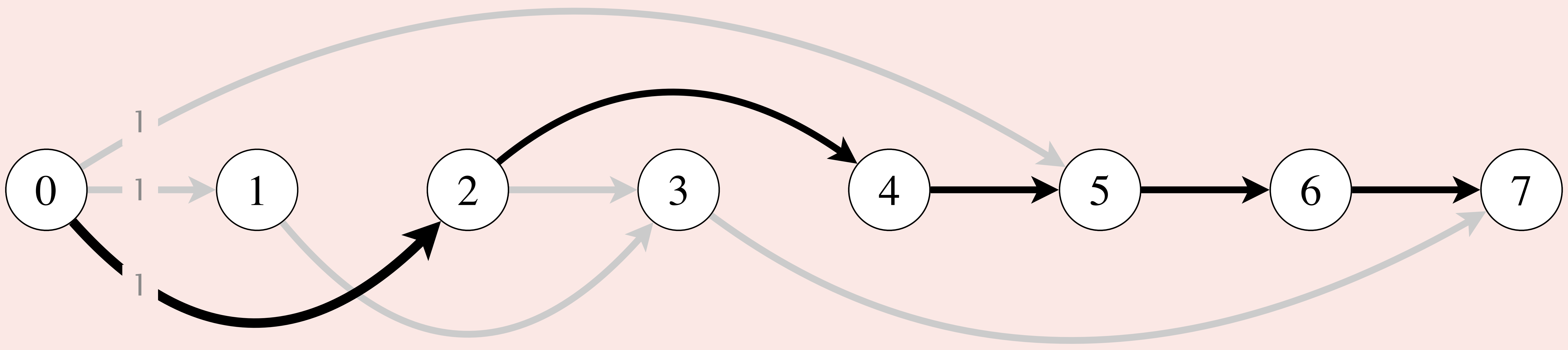
- Traceback: `distTo[v] == distTo[u] + e.weight()`.
- Or, maintain `edgeTo[]` array, as in Dijkstra / Bellman–Ford.




Equivalent (but simpler) computation. Relax vertices in topological order.

```
Topological topological = new Topological(G);
for (int v : topological.order())
   for (DirectedEdge e : G.adj(v))
      relax(e);
```

**How to efficiently find longest path from s to every other vertex in a DAG?**




longest paths problem in a DAG (all weighs = 1)

**A.** Negate edge weights and use DP algorithm to find shortest paths.

B. Replace min with max in DP recurrence.

**C.** Either A or B.

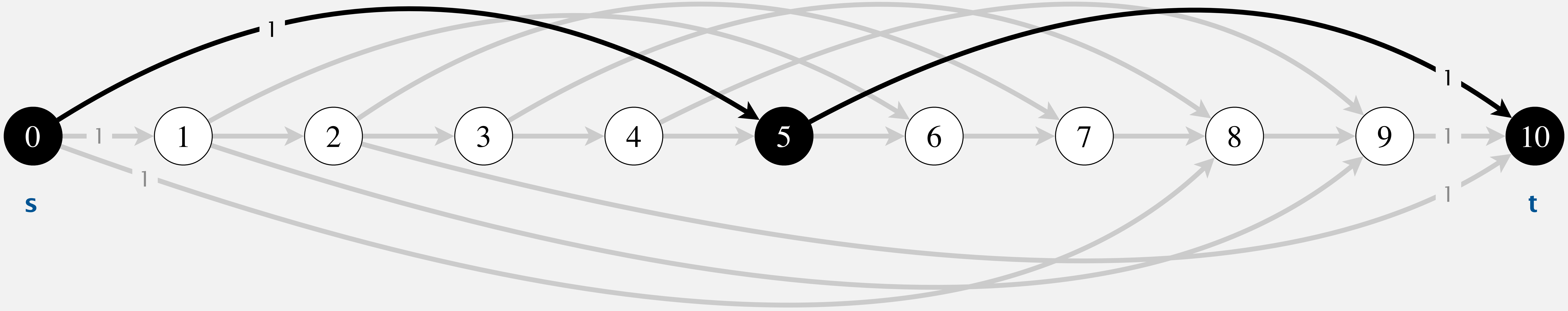
**D.** No poly-time algorithm is known (NP-complete).

# Shortest paths in DAGs and dynamic programming

DP subproblem dependency digraph.

- Vertex $v$ for each subproblem $v$.
- Edge $v \rightarrow w$, if subproblem $w$ depends on subproblem $v$.
- Digraph must be a DAG. Why?

Ex 1. Modeling the coin changing problem as a shortest path problem in a DAG.



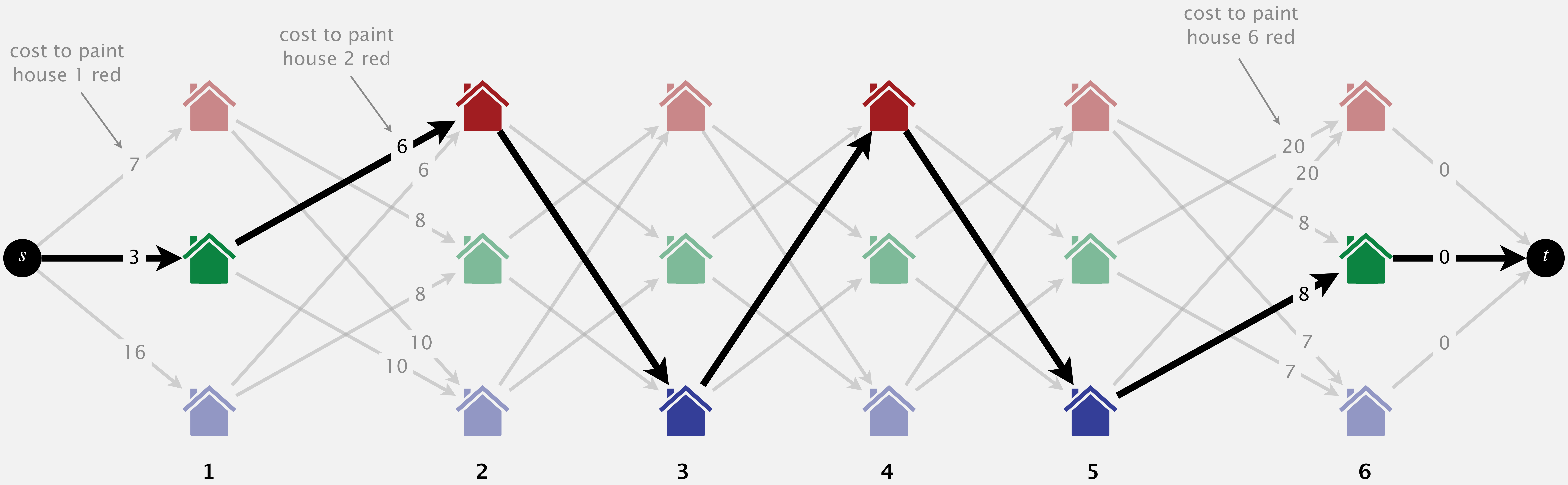

**V = 10; coin denominations = { 1, 5, 8 }**

# Shortest paths in DAGs and dynamic programming

DP subproblem dependency digraph.

- Vertex $v$ for each subproblem $v$.
- Edge $v \to w$, if subproblem $w$ depends on subproblem $v$.
- Digraph must be a DAG. Why?

Ex 2. Modeling the house paining problem as a shortest path problem in a DAG.

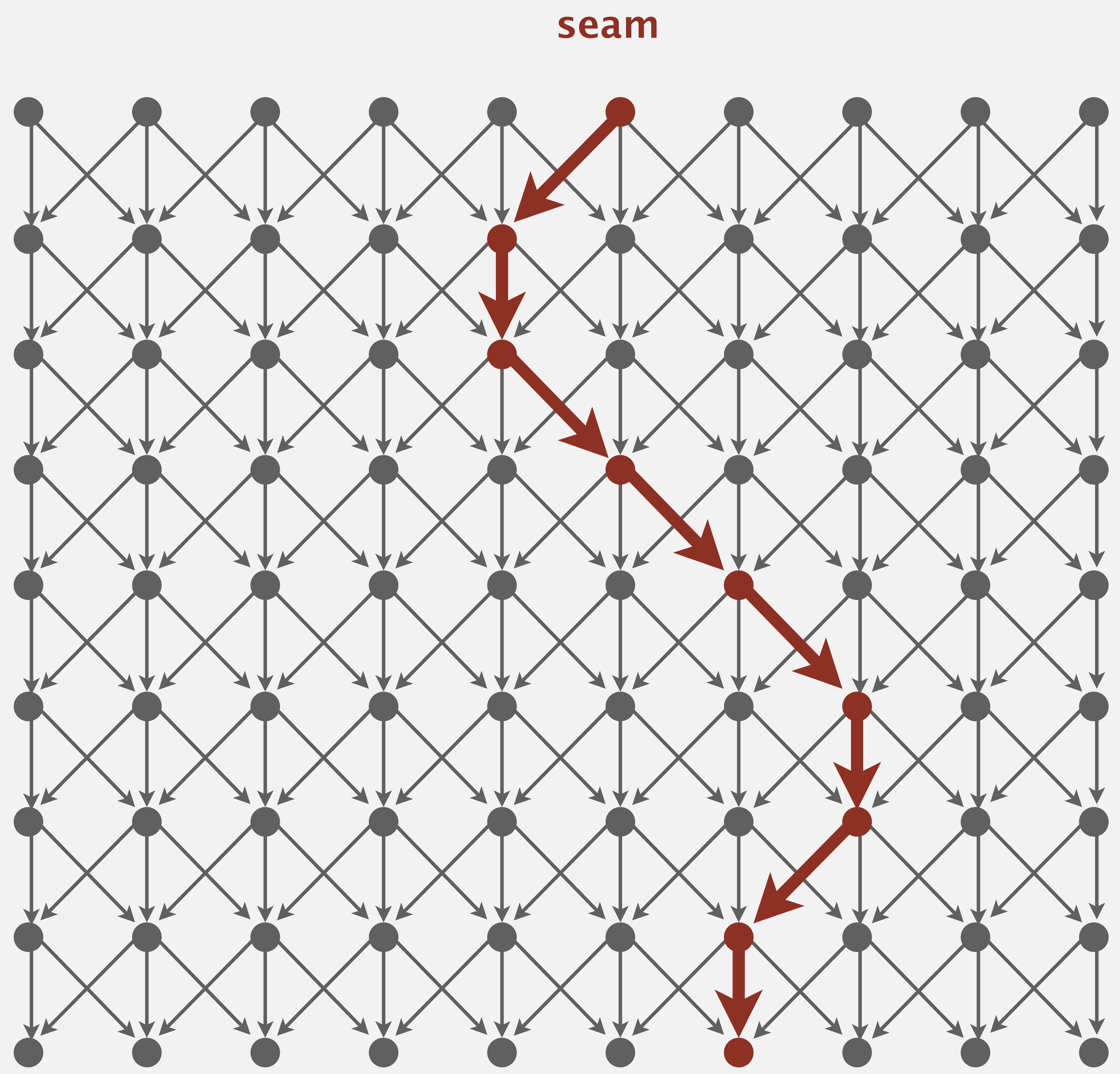


# Seam carving

Problem. Find a min energy path from top to bottom.

Subproblems. $distTo(col,\ row)$ = energy of min energy path from any top pixel to pixel $(col, row)$.

Goal. $\min \{ distTo(col, H{-}1) \}$.



Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

# Dynamic Programming

- introduction
- Fibonacci numbers
- interview problems
- shortest paths in DAGs
- *shortest paths in digraphs*

# Dynamic programming: quiz 4

**Let G be an arbitrary digraph with positive edge weights.**

**Consider the following DP recurrence:**

$$distTo(v) \; = \; \begin{cases} 0 & \text{if } \; v = s \\ \\ \displaystyle\min_{e \, = \, u \to v} \; \{ \, distTo(u) \; + \; weight(e) \, \} & \text{if } \; v \neq s \end{cases}$$

**Why does it not lead to an efficient algorithm for the shortest paths problem?**

**A.** Invalid recurrence.

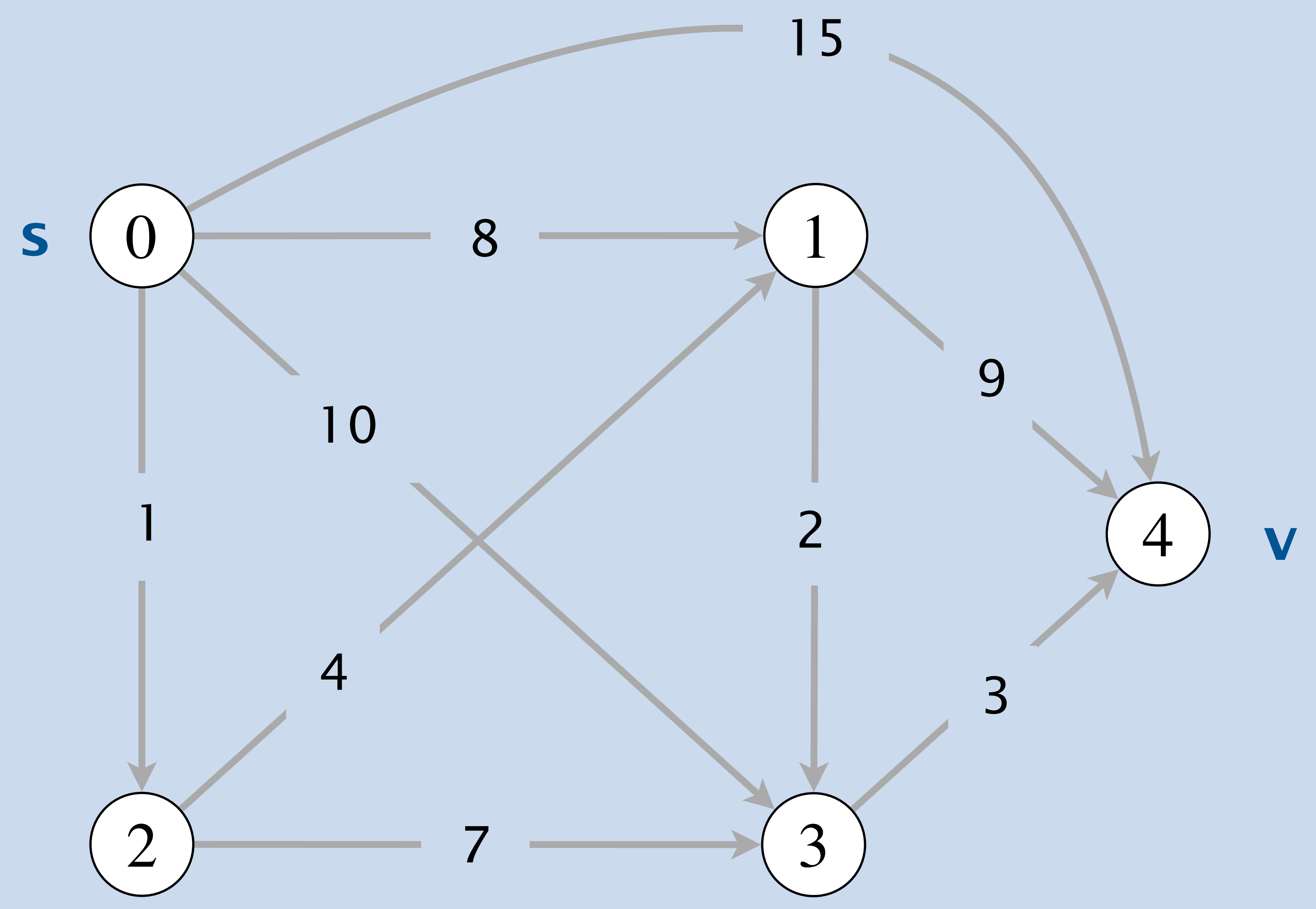
**B.** Leads to an exponential-time algorithm.

**C.** Need order in which to solve subproblems.

**D.** It does and algorithm takes $\Theta(E + V)$ time.

# SHORT SHORTEST PATHS

Goal. Given a digraph $G$ with positive edge weights and a source vertex $s$, find a shortest path from $s$ to each vertex $v$ that uses $\leq k$ edges.




k = 0: (∞)

k = 1: s→v (15)

k = 2: s→3→v (13)

k = 3: s→2→3→v (11)

k = 4: s→2→1→3→v (10)

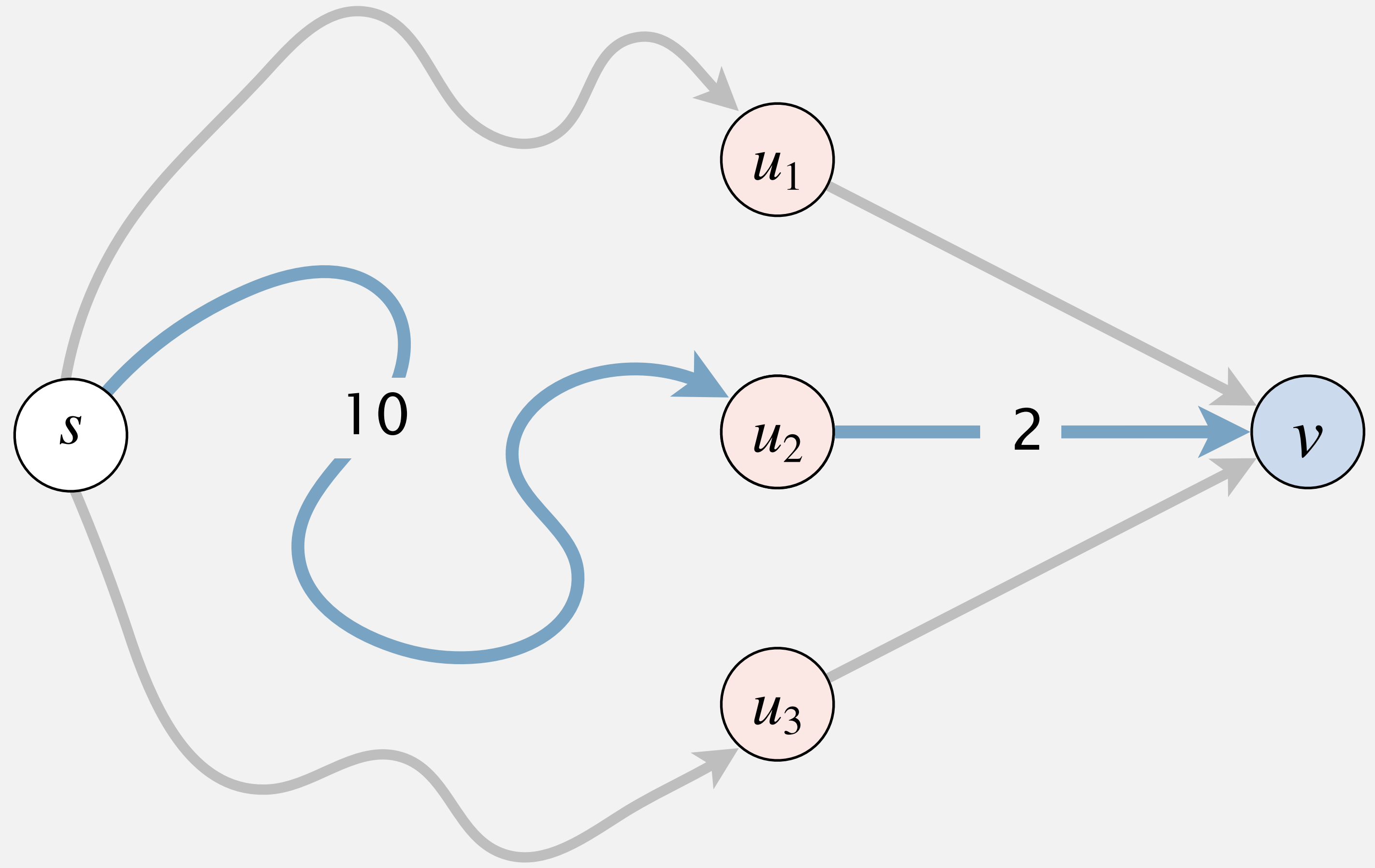
k = 5: s→2→1→3→v (10)

# Short shortest paths in digraphs: dynamic programming formulation

**Problem.** Length of shortest $s \leadsto v$ path that uses $\leq k$ edges.

**Subproblems.** $distTo(v, i)$ = length of shortest $s \leadsto v$ path that uses $\leq i$ edges.

**Goal.** $distTo(v, k)$ for each vertex $v$.




**Multiway choice.** To compute $distTo(v, i)$ :

- Select an edge $e = u \to v$ entering $v$.
- Combine with shortest $s \leadsto u$ path that uses $\leq i - 1$ edges.

take best

optimal substructure

**Dynamic programming recurrence.**

$$distTo(v, i) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min\limits_{e = u \to v} \{\, distTo(u, i-1) + weight(e) \,\} & \text{if } i > 0 \end{cases}$$

# Dynamic programming: quiz 5

**In which order to compute distTo(v, i) ?**

**A.** Increasing $i$, then $v$.

**B.** Increasing $v$, then $i$.

**C.** Either A or B.

**D.** Neither A nor B.

```
for (int i = 1; i <= k; i++)
   for (int v = 0; v < G.V(); v++)
      distTo[v][i] = ...
```

```
for (int v = 0; v < G.V(); v++)
   for (int i = 1; i <= k; i++)
      distTo[v][i] = ...
```

$$distTo(v,\, i) \;=\; \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min\limits_{e = u \to v} \{\, distTo(u,\, i-1) + \; weight(e) \;\} & \text{if } i > 0 \end{cases}$$

# Short shortest paths in digraphs: properties

Running time. DP algorithm takes $\Theta(k\,E + V)$ time.

Easy to modify DP algorithm to find the shortest path itself (not just the length).

- Approach 1: traceback.
- Approach 2: maintain `edgeTo[v][i]` entries along with `distTo[v][i]`.

# Shortest paths: Bellman–Ford vs. dynamic programming

DP algorithm can be used to solve single-source shortest paths problem.

- Choose $k = V - 1$. ← since weights are positive, shortest path uses $\leq V - 1$ edges
- Takes $\Theta(E\,V)$ time and uses $\Theta(E\,V)$ extra space.

Bellman–Ford can be viewed as DP algorithm, plus a few optimizations.

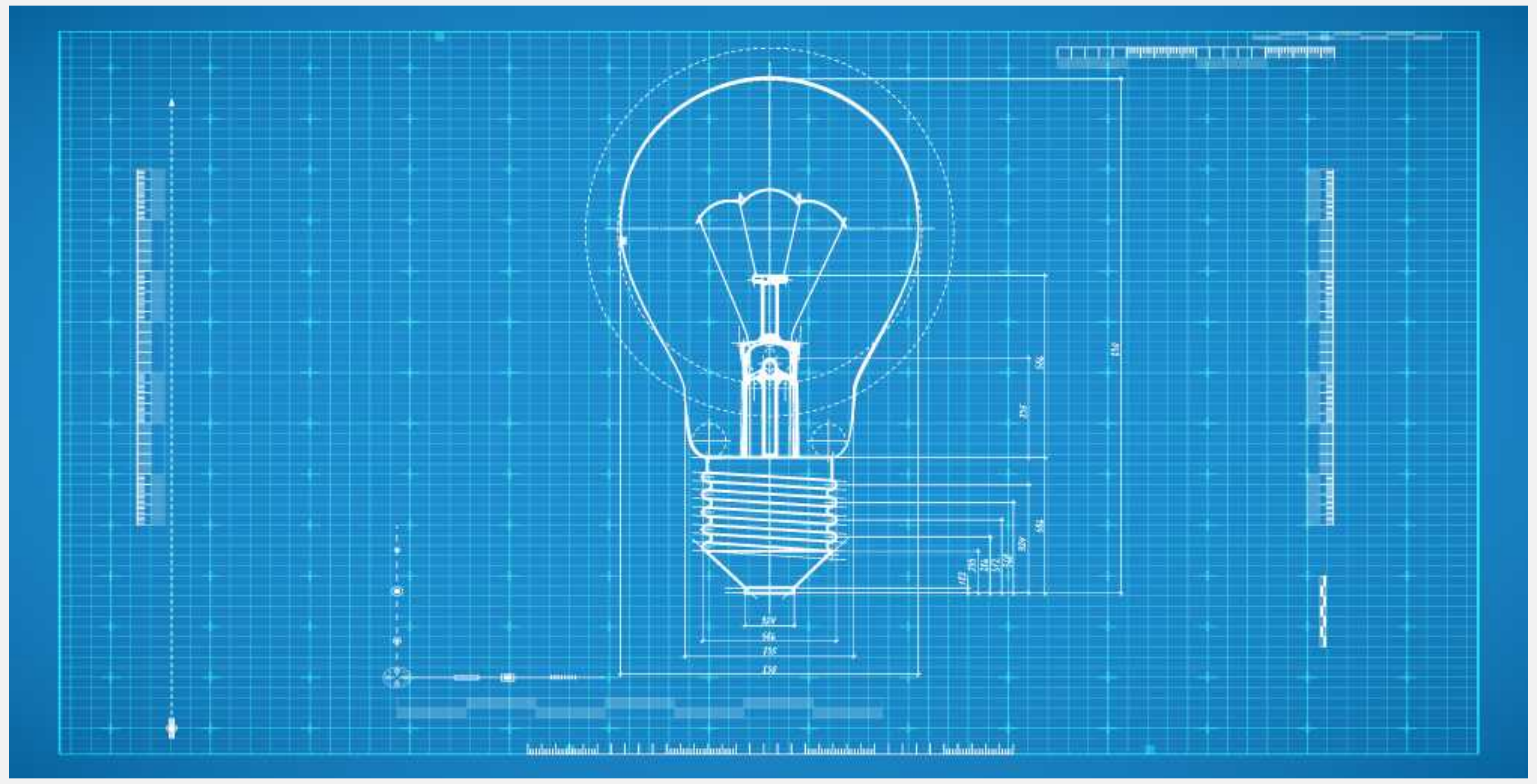
- Space optimization: uses a one-dimensional array `distTo[]`.
- Reorders computation: relaxes all edges incident from $v$.
- Performance optimization: uses a queue to avoid unnecessary work.
- Takes $O(E\,V)$ time and uses $\Theta(V)$ extra space.

can be much faster than $\Theta(E\,V)$ in practice

# Summary

How to design a dynamic programming algorithm.

- Find good subproblems. 
- Develop DP recurrence.
  - optimal substructure
  - overlapping subproblems
- Determine order in which to solve subproblems.
- Cache computed results to avoid unnecessary re-computation.
- Reconstruct the solution: backtrace or save extra state.



© Copyright 2020 Robert Sedgewick and Kevin Wayne